SICS/T-90/9003

MUSE TRACE
A Graphic Tracer for Or-parallel Prolog
by

Claes Svensson and Jan Sundberg

SICS technical report
T90003
ISSN 1100-3154

MuseTrace

A graphic tracer for Or-parallel Prolog

Claes Svensson & Jan Sundberg

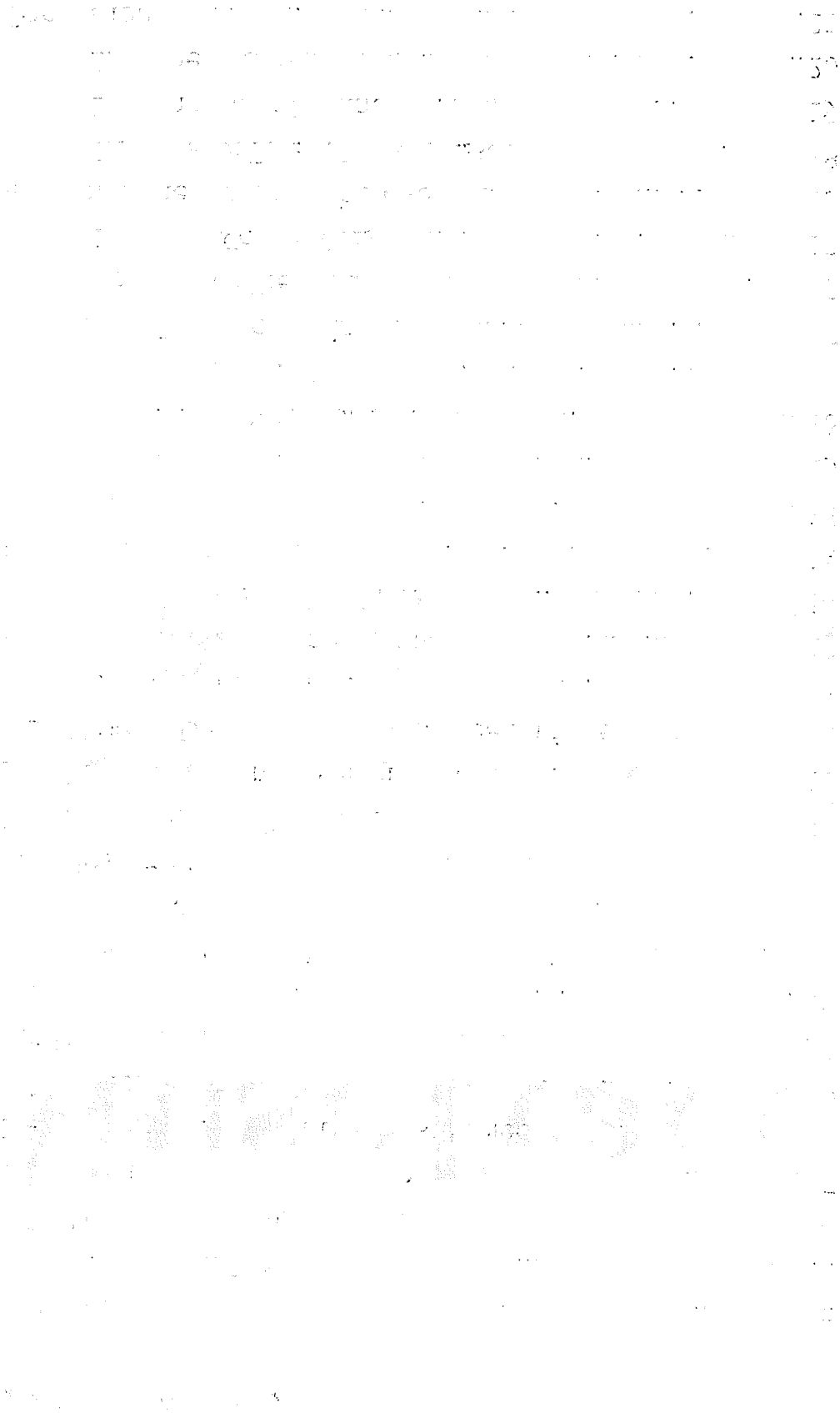


Table of Contents

Abstract

Must is a graphical tracer for the or-parallel Prolog language Muse. Its purpose is to aid development and evaluation of Muse. The program can also be used for demonstration purposes. Must runs under UNIX and X-windows. Our goal was to make it simple to use but still expressive. This report will describe the background, design, and implementation of Must. It will be concluded by an evaluation of the Must impact on the development of Muse algorithms. The report will also include a description of Stat, a tool for evaluating potential Or-parallelism in Prolog programs.

Acknowledgements

We would primarily like to thank Roland Karlsson and Khayri A. M. Ali, the two most important persons in the Muse project, for helping and leading us through our development of Must. Roland Karlsson for the development of the Muse tracing mechanism and Khayri for the theoretical aspects in the same area. We would also like to thank Johan Widén for helping us in choosing the right graphic environment and development tool, Interviews. He also helped us through early versions of the used compiler and Interviews.

1 Introduction

In this section we describe the background of Must, including an introduction to Or-parallel Prolog and the Muse approach. We also explain the goals of Must and give a brief history of the project.

1.1 Or-Parallel Prolog

The two main kinds of parallelism in logic programs are And-parallelism and Or-parallelism [1]. Or-parallelism allows alternative solutions to be tried simultaneously. And-parallelism allows conjunctions of goals to be tried simultaneously. Since the proclamation of the japanese fifth generation project much research has been done in combining parallelism and logic programming. The major advantages of Or-parallelism are simplicity and generality. The language could be used to execute a wide range of applications effectively since Or-parallelism occurs in many areas, including expert systems, theorem provers, natural language applications and data bases.

The main problems would be minimizing the overhead of binding and unbinding variables, garbage collection, load balancing etc. Remember that one of the most important goals of the parallel language is to be faster than any sequential Prolog in a large group of application. Else it is hardly motivated at all. Experience shows that it is indeed possible to implement a fast and effective or-parallel prolog systems.

1.2 The Muse Approach

The origin of the Muse approach is a research project started at SICS in 1986 called the BC-machine[2]. The purpose of the project was to find an approach for or-parallel execution of Prolog on multiprocessors with distributed memory. The idea was to assume a number of sequential Prolog engines and some shared memory for control information in the system and reduce the copying overhead with a special broadcast network for parallel copying. Actually the copying overhead was discovered to be a lesser problem than expected. So the approach was refined and became suitable for any multiprocessor system supporting local and global address spaces and copying, from one local memory to another [4].

Muse syntax is very similar to the sequential SICStus prolog [3], the difference mainly being the declaration of which predicates to run in parallel, and which to run sequentially [4]. Muse is currently implemented on a 7-processor distributed memory prototype constructed at SICS consisting of 7 68020 CPU's, 7 2.5 MBytes local and 4 MBytes global memory and a common VME-bus. It is also implemented on a Sequent Symmetry shared memory multiprocessor system with 16 processors. It has also been ported to Sun3 and Sun4 UNIX uniprocessor workstations [4].

The current implementations of Muse, so far, support Commit Prolog [4], a variant of Prolog using cavalier commit instead of cut and parallel side-effects instead of sequential [4]. Cavalier Commit prunes the branches on both sides of the committed branch, and is not guaranteed to prevent side effects from occurring on the pruned branches [6]. To obtain full Prolog semantics, it is necessary to follow certain rules that restrict the degree of Or-parallelism for programs with cuts and sequential side effects. The next step is to support cut and sequential side effects without restricting the parallelism. The results achieved with Muse are very promising, with circa 5 % extra overhead compared to standard Sicstus Prolog per processor and a speedup factor close to the number of processors in the system for a large class of problems.

Some terms that will be used later need to be explained here:

- A *parallel node* is a node in the Prolog search tree whose branches can be executed in parallel, i.e. any number of branches can be executed simultaneously.
- A *sequential node* is a node in the Prolog search tree whose branches must be executed sequentially, i.e. one at a time.
- A *worker* is an independent processing unit working in the Prolog search tree. This could be a processor or just a process depending on the implementation.
- The *shared* part of the tree is the part of the Prolog search tree that more than one worker have in common.
- The *unshared* parts of the tree are the parts that the workers haven't shared with each other.
- *Local load* is a measure of how much work a particular worker has got in his unshared part of the tree.

1.3 The Purpose and Goals of Must

The inspiration for Must was a graphical tracer, called WamTrace [7], defined for Aurora [5], another Or-parallel Prolog implementation. WamTrace takes a dump of an Aurora execution and shows an animated display of the search tree and what the workers were doing through the trace [5].

Must uses some ideas from WamTrace but enhances the ideas notably and adds quite a bit of its own. The purpose of the program is to give an animated tracing of a parallel execution of a Prolog query. The tracing should include a display of the search tree that clearly and intuitively describes how the real search tree looked like during execution. It should also show what each worker (processor or process) is doing at any time. You would probably also want some statistics shown, like how busy the workers are as a function of time. Later in the development we also included printing-facilities and lots of other functions.

Two of the main differences between Must and WamTrace is that Must traces real time information and the utilization of workers in the system. This is extremely essential for understanding whether the language implementation is correct or not. Must also traces on query level, compared to WamTrace that traces a complete Prolog session. Even if the Prolog program should crash during execution of a query, Must will trace what happened before the crash.

Our goals in constructing Must can be summarized as follows:

The Must program should be

- *easy to use* for anyone,
- *expressive*, showing all the information in an intuitive way,
- *easy to expand/change*, to show any additional information found to be needed after the program is completed, and
- general enough to be used for *multiple purposes* : debugging, evaluation, demonstration etc.

Further on in this paper we will give an account on how and in what degree we achieved these goals.

1.4 A Short History of the Must Project

The Must project was suggested to us as a Master of Science thesis in summer 1989 by Khayri Ali and Roland Karlsson. The project was to be sponsored by SICS. We began to work on it in early October and finished in late December. The first two weeks or so were spent getting acquainted to the environment and deciding on some of the design issues mentioned further on in this report. The rest of the time was dedicated to developing and documenting Must. The features of Must were changed several times during this time as new requirements were found and as new ideas popped up from the Muse-people.

1.5 Structure of the Paper

We have organized the paper as follows. Section 2 describes the environment we worked in. Section 3 describes the major design decisions made while planning and implementing Must. Section 4 describes the implementation and structure of the program. Section 5 consists of a conclusion and an evaluation of the impact Must has had on the development of Muse. In the appendices you will find a user manual to Must and a description of Stat, a tool developed by us for evaluating potential Or-parallelism in Prolog programs.

2 Environments

In this section we describe the environment choices made while developing Must, the main issues being X-Windows, Object-Oriented Programming, C++, InterViews and PostScript.

2.1 X-Windows

The program was specified to be able to run on Sun workstations under UNIX. Knowing that the program would use quite a bit of graphics we looked at X-Windows and SunView. We found X-Windows to be a very powerful and general environment to work with [8]. With X-windows you can run the program on any UNIX-machine and show the results on any display supporting the X model, for example on Sun workstations, X-terminals etc. The problem was only that the X-interface seemed quite messy to work with. We examined the code written for X-Windows in Wamtrace [7] and didn't like what we saw. This problem was soon to be solved though, as will be explained below.

2.2 Object-Oriented Programming and C++

The advantages of object-oriented programming have been widely discussed in both the academical and the professional worlds [9]. The main idea is to work with objects, a kind of data structures consisting of collections of data and attached routines called methods. Each object belongs to a class, a type definition that defines what an object contains and what methods are associated with it. The classes belong to a hierarchy that defines how the classes inherit from each other. You say that if a class A inherits from class B, then A is a subclass of B and B is a superclass of A. Some examples of object-oriented languages are Smalltalk, C++, Objective C, Modula 3 and a variety of Lisp-variants. There is even object-oriented Prolog. "Object-oriented" has become something of a buzz-word in the eighties just like structured programming was in the seventies. It's fashionable to be object-oriented. But it isn't just a fashion that will be gone soon. Object-oriented programming is probably here to stay. It's very nice to program in an object-oriented language and once you've gotten used to it, it's hard to go back to standard C or Pascal.

As we had had contact with the object-oriented approach earlier, we soon found that it would be very well suited to the kind of program we were contemplating. The choice of object-oriented languages available in this environment wasn't large though. The only one good enough was C++. C++ isn't a very strict object-oriented language [10]. It's as dirty as C but with some niceties of its own as well. The advantage of C++ is that you've got lots of power and freedom in your hands if you're careful and know what you're doing.

The main drawback in this case was that the C++-compiler wasn't completely debugged and had some strange peculiarities. While we were writing the program we got newer releases that fixed some of the problems, but created some new ones instead. The main reason why we chose C++ was that we had found a C++-library called InterViews that turned out to be a great help for us. InterViews will be described in the next section.

Our goal to make Must easy to expand was achieved almost for free when we chose to work in an object-oriented language. The modularity supported by object-oriented programming makes this very easy to achieve. After a few hours of source code demonstrations, our tutor was able to make all the extensions he wanted without problems.

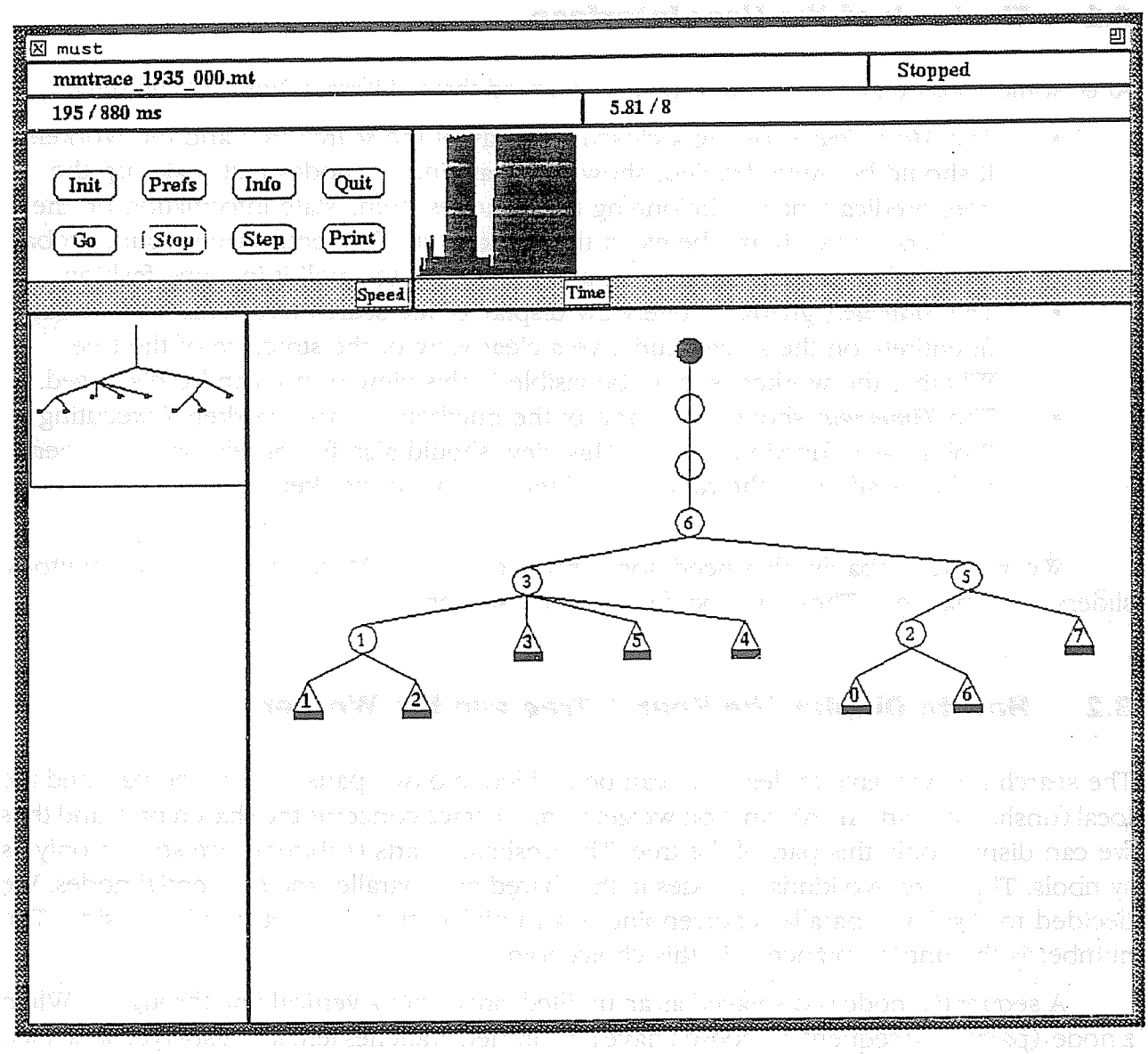
2.3 InterViews

InterViews is a toolkit for building and administrating user interfaces, structured graphics and many other useful things [11]. It's written in C++ and is used under X-Windows. Using InterViews, you never have to touch the X-Windows interface directly at all. InterViews has got classes that implement windows, menus and buttons of different varieties, handle user input from mouse, keyboard etc, interprocess communication, text, graphics and much more. All you have to do is to create your own subclasses to the InterViews classes so you get exactly what you want. The drawback is that InterViews defines a certain kind of interface and if you want something radically different, you might have a hard time telling InterViews what to do. In the implementation section we show how we have built our own subclasses from the InterViews classes. These classes handle the user interface and graphics.

To implement the data structure (the search tree etc), we had to create entirely new classes of course, as these have nothing to do with the user interface. Our experience of InterViews give us a distinct feeling that this kind of toolkit should and will be extensively used in the future to build interactive applications. We urge all developers of applications with interactive user interfaces and graphics to examine the possibilities of using InterViews or something similar. There will probably appear more products on this market in the near future inspired by the successes of similar products on the personal computer market. Especially we find the combination of object-oriented toolkits and a standardized window manager very promising.

2.4 PostScript

The need for a printing facility became evident at a very late point in the development. We found that the easiest way to get high-quality graphic printouts, was by generating PostScript code and sending it to a laserwriter [12]. The things we wanted to print was the tree and the busy-graph (the utilazation of processors over time). The code for both the tree and the graph can be generated with the data structure already present (see section 4). All we had to do was write methods for the apropriate classes that, using the data already present, generate the PostScript code. This code is written to a file together with code that do scaling etc. This file is then sent to the laserwriter with the shell command lpr. Lpr uses the default printer, so we also put in the possibility to just create the PostScript file called PSdump. The user can then send the file to any printer, save it or manipulate it in any way he wants.



The user interface

3 Design of the User Interface.

The design of the user interface includes several important questions:

- *What* information do we want to see?
- *How* can this information best be displayed?
- In what way should the user be able to *control* the display?
- How can we keep this *simple* and still be able to show everything we want?

3.1 The Look of the User Interface

After some consideration we found that we wanted three different views of the trace:

- The *Mainview*, showing a closeup display of the search tree and the workers. It should be quite detailed, showing what kinds of nodes that make up the tree, predicate names belonging to the nodes, some state information on the workers, some signals between the workers etc. The entire view could probably not fit on the screen, so we should be able to scroll it in some fashion.
- The *Sideview*, giving an overview display of the search tree. This view should fit entirely on the screen and give a clear view of the structure of the tree. Whether the workers should be visible in this view or not can be discussed.
- The *Timeview*, showing a graph of the numbers of busy workers (executing Prolog) as a function of time. This view should also fit entirely on the screen and thus adapt to the runtime and the number of workers.

We would probably also need some ways to control Must, including pushbuttons, sliders, scrollbars etc. These will be discussed in section 3.3.

3.2 How to Display the Search Tree and the Workers

The search tree we have to deal with can be divided into two parts: the shared part and the local (unshared) part. All information we get from the trace concerns the shared part, and thus we can display only this part of the tree. The unshared parts (subtrees) are shown only as symbols. There are two kinds of nodes in the shared part: parallel and sequential nodes. We decided to display a parallel choicepoint as an unfilled circle with a number inside. The number is the untried branches in this choicepoint.

A sequential node is displayed as an unfilled circle with a vertical line through it. When a node (parallel or sequential) doesn't have any untried branches left, it is displayed as a filled circle. The tree is displayed with the root at the top, growing downwards. Whether to do it this way or the other way around is not entirely obvious, but it looks nicer and is more intuitive (at least to us). Each choicepoint is connected to its father with a thin line.

An unshared part of the tree is displayed as a triangle or as a rotated rectangle depending on the state of the worker that handles this part of the tree. The workers are shown simply with a character, counting from zero and up.With more than ten workers we continue with A through Z. We had many ideas of displaying the worker in a much more complicated fashion, as some sort of icon that would change with the state of the worker. We abandoned these as we found that the clarity of a simpler approach greatly outweighed the advantages of this complexity. The worker is displayed next to the choicepoint it is currently working on, except when it is in an unshared part of the tree. Then it is moved inside the triangle/square. We use the rotated square symbol to show that a worker is not executing Prolog but copying, signalling etc. A filled bar is drawn under the triangle when there's untried branches that the worker can share with an idle worker.

If we want to show signalling between the workers, we decided to display this as a beam from the signalling worker to the signalled. The signals to be showed were decided to be share-requests and commit/cut-signals. We also display the commit/cut operation with a beam from the committing worker to the committed choicepoint, followed by the appearance of a pair of scissors across this choicepoint. Notice that we might not always want to see all signals. They can be confusing and should be able to turn off.

When considering the sideview, it became obvious to do some scaling of the view as the tree grows. The tree should also be much less detailed than in the mainview. We decided to only show the connections between each choicepoint and its parent. This is enough to show the structure of the tree. After further consideration we also included the workers, shown as small dots in the tree. The tree is shrunk to two thirds of its vertical size each time it hits the bottom. The horizontal size is never a problem (assuming no suspended branches) as the tree, both in the mainview and the sideview, is continuously adjusted to fit the window.

3.3 How to control Must

The main questions here are:

- *What* should be controlled?
- *How* should this be controlled?

There are several things you would like to control as a user. They can be divided into two main categories: display- and flow control.

3.3.1 Control of the Display

There are two things you want to control in the display. What part of the tree to show in the mainview and how much detail to show there. What part of a view to show is normally controlled with a scrollbar. We have a scrollbar to, but it's kind of special. We use the sideview as a scrollbar. A rectangle is drawn in the sideview, showing exactly what part of the tree that the mainview depicts. To show another part of the tree, the user simply grabs hold of the rectangle and drags it to proper place, and the mainview scrolls simultaneously. You can also simply click in the sideview on the part of the tree you want to see, and the mainview shows this part immediately. This kind of "graphic scrollbar" is a very nice feature, that might find use in other applications, like layout-programs etc.

To control how much to show, we chose to use a pushbutton labelled "Prefs" that pops up a dialog where the user can choose whether to show signals, predicate names (labels), sliding workers or not. Showing these extra features slows the program down and makes the display more complicated. As an option, all the trace primitives can be written out in text on stdout, as they are executed. This option is set by choosing "Debug" in the "Prefs"-dialog. This way you can get all information in the tracefile in a readable format.

3.3.2 Control of the Trace

We decided to see the trace like a tape recording. The user should be able to start, stop, rewind etc. We use pushbuttons to "Stop" and "Go". To skip to another place in the trace, we have a slider labelled "Time". The user drags the rectangle to the place he wants to go, and the program quickly skips to this place without showing anything. If the user wants to "step" through the trace he can use the button labelled "Step", that advances the trace the smallest step that would cause any changes in the display. This means that if the trace includes a lot of information without graphical consequences, the program keeps running until something visible happens. Besides single-stepping Must has another useful stepping-feature. By pushing the left and right buttons on the pointing-device (mouse) in the time-slider, the program skips a small amount backwards or forwards. The amount is 1, 10, 100 or 1000 trace primitives, chosen by the user in the "Prefs"-dialog.

To control the speed of the trace we have a slider labelled "Speed". We also have an "Init"-button that restarts the trace from the beginning. To "insert a new tape", that is start to work on another file, the user clicks on a field labelled with the trace's name. A dialog with a list of your directory pops up, so you can choose a new tracefile. If a directory is chosen, a new list is shown of the files in this directory. We decided that the user also should be able to choose tracefile or a directory to start from when he starts Must. The argument to Must is interpreted as a tracefile or else as a directory to start working in.

As a conclusion, the user interface is simple to use but still quite powerful. Of course, the interface is custom-made for the Muse-developers and might not suit other users exactly that well, but we think that the principles we have used are quite general. The user interface should be easy to use for anyone that is previously acquainted with interactive user interfaces, given some training. We think that almost all applications will have interactive graphic interfaces in the near future. The lack of standards might become a problem, but it will probably be solved.

4 Implementation

Building the Must program, we soon discovered that the things to do during execution could be divided into two strictly independent parts: manipulation of the data structure and the corresponding actions to make them visible in the user interface. This made us split the implementation of Must into the same two parts, i e the development of an appropriate inner data structure and the building of the user interface.

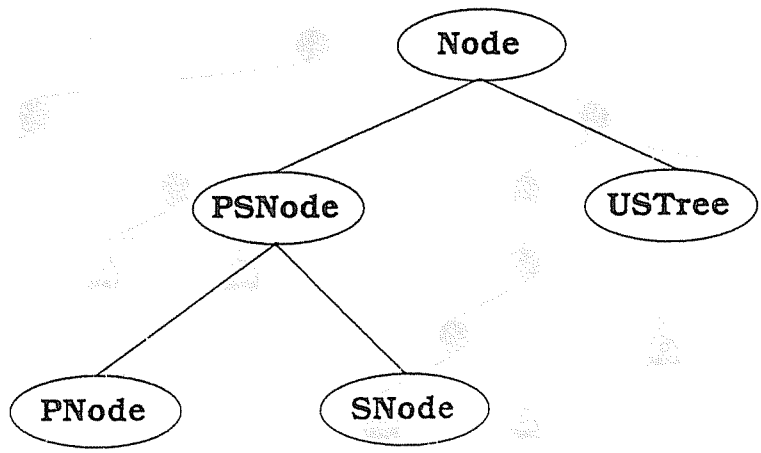
The first sections describe the data objects that form the Must data structure, the structure and behavior of these objects and the relationship between them. In the sections following them we give the same treatment to the user interface. As mentioned before, Must is written entirely in the object oriented programming language C++. This means that the data structure and the user interface elements are objects created from classes. The object are referred to as instances of the class names.

4.1 The Data Structure

There are only two types of data objects forming the Must data structure: the node objects and the worker objects. They represent the Prolog system search tree, all the workers in the tree and what they are doing to the data structure.

4.1.1 The Node Class

One of the two fundamental classes in the Must data structure is the node class. It is the base class in a hierarchy of classes, and contains fields and methods that all node classes have in common. The relation between the different node classes is showed in the figure below.



The relations of the node classes

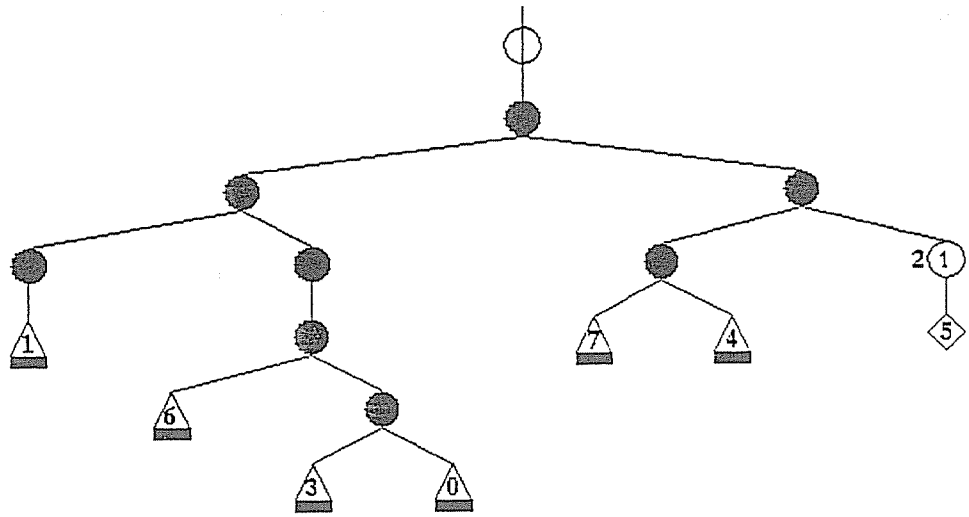
The kind of data all the node classes have in common are (among others not needed to mention here) :

- references to a list of the workers assigned to it,
- references to the parent node and the left and right sibling nodes in the data structure,
- graphical objects visible in the user interface, like the node circle and a connecting line to its parent node, and
- a coordinate specifying the y offset in the tree view of the user interface.

The methods in common are used for adding sibling nodes to the same level in the search tree (both in the data structure and the user interface). The node class is never instantiated in the data structure, it is only used as a generic class.

There are four more fields in the Node class that require a more specific explanation, namely the *innode*, *undernode*, *offset* and *width* fields. These are used to "balance" the tree during execution, i e see to that the nodes in the user interface are positioned correctly in the view with respect to the tree structure. The width field specifies the total width of the reserved view space for the node while the offset field specifies the screen coordinate where that view space begins. To make it easier: the x coordinate of a node is offset + width/2 along the x axis, i e in the middle of the reserved view space of the node.

In our first approach to an algorithm for keeping the tree balanced, we divided the width space of the parent node equally among its children. We soon discovered that with this algorithm, the available space in the tree view soon was "consumed", especially with deep and dense tree structures like self similar trees. A more efficient way to share the view space along the x axis is to do it with respect to the workers current positions instead. This is where the innode and undernode fields come in, specifying the number of workers at and somewhere under the node. The view space of the parent node is now divided among its children, each child getting its weighted share of user space, based on the number of workers at and under the node. This algorithm uses the view space in the user interface in a much more efficient manner and displays deep and dense tree structures very clearly. The figure below should help you get the picture.



A balanced tree example

4.1.2 The PSNode Class

The PSNode class is also a generic class, used to generate the classes for sequential and parallel nodes in the data structure. Here we find fields like the number of branches left to try, a reference to the leftmost child of the node, a string with the predicate name and a boolean value indicating if the node is marked as committed. We also find methods for creating a new child and for aligning the nodes in the user interface the correct way depending on the number of workers on the current and underlaying levels (explained later).

4.1.3 The SNode Class

The SNode class inherits directly from the PSNode class and is used to represent sequential nodes in the data structure, i e the nodes that execute their branches one at the time. They are represented in the user interface by a circle with a vertical line through it. The workers (processors or processes), if any are present, are aligned to the left and, if requested, the predicate name is aligned to the right (see the Must user manual).

4.1.4 The PNode Class

The other class that inherits from the PSNode class is the PNode class, which corresponds to parallel nodes in the data structure. Parallel nodes are the nodes whose branches are executed in parallell. They have a graphical representation in the user interface consisting of a circle with a number inside it telling how many branches there are left to try. If there are no more branches left to try the node circle is filled with black. Like the sequential node, it has the workers to the left and the optional predicate name to the right. The only thing that makes it different from the SNode is the graphic representation of it and thereby it has to be supported with its own allocation, manipulation and printing methods.

4.1.5 The USTree Class

The USTree class is used to implement local, unshared trees in the data structure and corresponds to the local workspace of a busy worker. Directly inheriting from the Node base class, it has almost nothing in common with parallel or sequential nodes. The most important thing to notice is that it has got only one field telling how many unprocessed branches there are left to process in the local search tree.

The graphical representation of an unshared tree is a little bit more complicated. An unshared tree with a worker in it is basically represented with a triangle with the worker number inside it. We know that there is a single worker since only one worker can be present in an unshared tree. As soon as the worker processing the unshared tree stops working (i e the worker mode changes to some non-busy mode) due to some request or other reason, the triangle is replaced with a rotated square. If the worker has more than 0 unprocessed branches (i e there is work left to be done), the symbol is marked with a small black rectangle underneath it.

4.1.6 The Worker Class

The other fundamental class building the Must data structure is the worker class which represents a worker. The worker objects in the data structure are to be seen as the "master objects" and are the ones that are "doing the real work". For every trace primitive read from the trace file in the main loop, a corresponding worker method is invoked. The worker method invoked then manipulates the different node objects in the data structure according to the action requested, using the existing node methods. It also makes sure that the correct object methods in the user interface are invoked if the node manipulation caused a visual change.

The data fields in a worker object are:

- its own identification number (ID),
- a mode field telling in which mode the worker is at the current time,
- a reference to the node where the worker is being positioned,
- a boolean field telling if the worker is currently working independently in an unshared tree, and
- the corresponding graphic object (the worker ID) visible in the user interface.

The worker is supplied with a number of methods, one for every possible trace primitive and some other help functions performed frequently. This is a good choice of implementation since the trace primitives read from the trace file only tells which worker currently is doing something and what he is doing (with a varying number of parameters). The only thing that has to be done is to invoke the corresponding worker method with the actual parameters.

4.2 Implementation of the User Interface

As mentioned earlier, the Must user interface is completely built with the Interviews class library. Interviews offers a number of classes intended to build user interfaces with. This makes it easier to build your own user interface and gives you more time to develop the graphic part in your specific application.

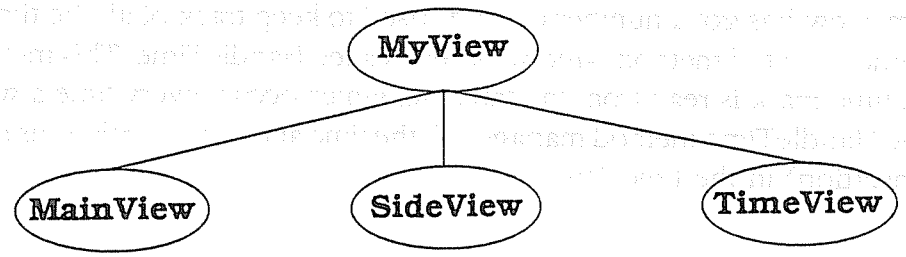
The Must user interface consists of many parts. Each part is an instance of a certain class, inherited from an Interviews base class. These parts can be divided into four categories: the views, the sliders, the buttons and the dialogs. We will only pay little attention to the button and the dialog class, they are to be seen as help classes. The following short description should be enough.

The button class allows you to associate a routine with a button in the user interface, which is invoked as soon as the button is pressed. The Must user interface is equipped with eight such buttons to control the Must execution. The dialog class lets you compose dialog boxes for explicit user interaction like data input and user settings changes. Must is equipped with three dialog boxes, one for choosing a trace file, one for trace file information and the last for setting the user preferences.

4.2.1 The Different View Classes

The views in the user interface are of two types, one for text messages and the other for continuously changing graphics. The first type - inheriting from an Interviews base class - is used for text fields in the user interface and has three instances in Must. They show the Must status, current run-time and speedup so far. The user interface is built in a way that makes it easy to add more text fields afterwards if necessary. Another almost identical class with one instance in Must is used for showing the current trace file and for choosing a file by popping up a file choosing dialog box when clicked upon. It differs slightly from the other class because it needs its own Handle method, necessary to catch the mouse clicking event when the user wants to choose another trace file.

The latter kind of views are a bit more complicated than the ones previously described. The three views of this type in the user interface are the mainview, the sideview and the timeview. They are all the only instance of three different view classes (MainView, SideView and TimeView class) inheriting from an Interviews base class, GraphicBlock, via a common generic class, called MyView. See the figure below.



The relationships between the view classes

The MyView class is the general view class supplying methods for adding, removing and aligning Graphic objects into the view. A Graphic object, supplied by Interviews, is almost any kind of graphic object, a circle, rectangle, bitmap or whatever. All the graphic changes in a view are handled by a MyView field, the Damage object. By invoking the Damage method Incur every time the view changes - and passing the added, deleted or aligned Graphic object as a parameter - the Damage object adds the enclosing area of the Graphic to a list of areas that has to be repaired. The damaged areas of the view are then refreshed when the Damage method Repair is invoked.

4.2.2 The View Instances

The visually biggest view object in the user interface is the instance of the MainView class, simply referred to as the mainview. The same notation will be used describing the other views. The mainview shows (a part of) the search tree with all its details, like workers, the node circles and the number of branches left to try in each parallel node. This is the most "intensive view" requiring the most graphic manipulation. Almost every trace event read from the trace file causes the mainview to change in some way. The MainView class is almost identical to the MyView class except for printing information fields and different allocation and resizing methods.

The view to the left of the mainview is the sideview. It shows the whole, scaled, search tree without any details. Only the connecting lines between the nodes are visible. Every change in the mainview node structure causes the same change in the sideview. The sideview is also equipped with a rectangle showing the visible part of the search tree in the mainview. This rectangle can be grabbed and dragged vertically with the mouse which makes the mainview scroll to the new position. This is implemented with a special Handle method catching mouse click and move events and causing the mainview to scroll continuously.

The sideview is also equipped with two fields, specifying the current magnification along the axes in the view and a maximum field, used to detect when the sideview tree gets too big along the vertical axis, i e hits the bottom, and needs to be reduced. This is done by scaling all the Graphic objects in the sideview by 2/3 along the vertical axis.

The last one of the user interface views is the timeview. It is used to show a graph of the number of busy workers. It is implemented as a view that is to be filled with 500 black rectangles, each 1/500 along the x axis representing a time quantum of the total Prolog execution time. If the number of busy workers changes frequently during a time quantum the mean value is calculated and, on the other hand, if the next time mark is more than one time quantum away the corresponding space is filled with identical rectangles.

The timeview has got a number of fields used to keep track of all the time and summation values, and a central method (among others) called HandleTime. This method is invoked every time a time mark is read from the tracefile, which occurs every time a worker changes its mode. The HandleTime method manages all the time and sum calculations and all drawing (rectangle insertion) in the timeview.

4.2.3 The Sliders

The user interface of Must is equipped with two sliders, one for the trace speed setting and the other one for showing and setting the current relative time in the Prolog execution. They are instances of the different classes MySlider and TimeSlider, both inherited from an InterViews base class called Interactor.

The MySlider class is rather simple. A Handle method catches the mouse click and drag events in the slider which causes an execution speed parameter to be set when the button is released. This parameter is used to insert suitable delays in the execution.

The TimeSlider class is a bit more interesting. It is a subclass of the MySlider class and has got its own Handle method and two "skipping" information fields. The Must program enters the "skipping" mode whenever there is a mouse drag event in the TimeSlider instance. This causes Must to execute to the requested time in a fast way, much faster than the original execution speed.

This is done by executing the current trace file without any graphic consequences, only the data structure is manipulated. No graphic object, like node circles or connecting lines, are created and no view manipulation occurs. When the execution has reached the desired time point of the Prolog execution, the skipping stops and all the graphic objects building the trees in the mainview and the sideview are created by checking the data structure.

If the left or right mouse button is pressed instead of dragging with the middle one, the execution skips a certain number of steps forward or backward. The number of steps is set in the preferences dialog box. If the user wants to skip backwards the execution restarts from the beginning of the trace file and skips to the specified location. It has to be done this way due to the fact that no execution history is saved.

5 Conclusion

We have presented Must, a graphical tool for studying the Muse Or-parallel Prolog system. The structure of Must is simple and easy to modify for further extensions. With help of Must, many implementation and performance bugs have been discovered and an efficient Muse scheduler has been developed. It has also been very informative for demonstrations.

References

[1] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. Dept. of Applied Mathematics & Computer Science, The Weizman Institute of Science, Rehovot, May 1989.

[2] Khayri A. M. Ali. Or-Parallel Execution of Prolog on BC-Machine. In Proceedings of the Fifth International Conference on Logic Programming, pages 1531 - 1545, MIT Press, August 1987.

[3] Mats Karlsson and Johan Wide´n. SICStus Prolog User's Manual. SICS Research Report, Swedish Institute of Computer Science, October 1988.

[4] Khayri A. M. Ali and Roland Karlsson. The Muse Or-Parallel Prolog Model and its Performance. SICS Research Report, Swedish Institute of Computer Science, 1990.

[5] Ewing Lusk, David D. H. Warren, Seif Haridi et al. The Aurora Or-Parallel Prolog System. SICS Research Report, Swedish Institute of Computer Science, 1987.

[6] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In Proceedings of the Fifth International Conference on Logic Programming, pages 1590 - 1605, MIT Press, August 1988.

[7] Terrence Disz and Ewing Lusk. A graphical tool for observing the behaviour of parallell logic programs. In Proceedings of the 1987 Symposium on Logic Programming, pages 46 - 53, 1987.

[8] Oliver Jones. Introduction to the X Window System. (Prentice Hall, 1989).

[9] Brad Cox. Object-Oriented Programming: An Evolutionary Approach. (Addison Wesley, 1986).

[10] Bjarne Stroustrup. The C++ Programming Language. (Addison Wesley, 1986).

[11] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces with InterViews. Stanford University.

[12] Adobe Systems Incorporated. Postscript Language Reference Manual. (Addison Wesley, 1985).

STAT

A parallelism evaluator for Or-parallel Prolog

Table of Contents

Abstract

Stat is a simple tool for evaluating potential parallelism in Prolog programs. It takes a sequential Muse tracefile as input and gives a data printout that can be used to decide which predicates that should be declared as parallel and which ones as sequential. This paper describes the background, design and implementation of Stat. It will also include a user manual and a conclusion.

1 Introduction

Stat was written by us as a complement to Must. Its purpose is to give some hints on which predicates that should be declared as parallel and which that should be declared as sequential. Normally a program is examined analytically and this program doesn't replace this step but rather confirms your results.

2 Background

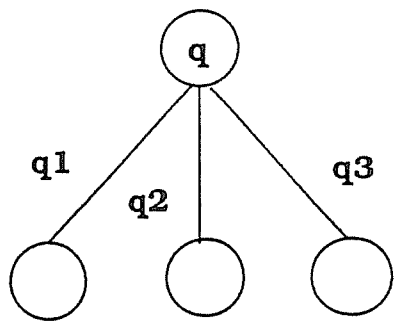
The big question here was: how do you find potential parallelism. What we have to work on is a sequential trace with basically four kinds of trace primitives:

- try, creating a choicepoint and taking the first branch,
- retry, taking the next branch of some choicepoint,
- remove, removing a choicepoint, and
- cut, selfexplanatory.

Try and retry give time information, so we can find out the execution time of each branch.

The basic idea goes like this:

Assume the tree in the figure below. For each choicepoint, q for example, we calculate tpar and tseq. Each branch has an execution time, let's call it ti. Tseq is the longest of these. Tpar is the sum of the rest. In the figure below, branch q1 takes t1, q2 takes t2 and q3 takes t3 to execute. Let's say that t1 is greater than t2 and t3. Then tseq is t1 and tpar is t2+t3. Running sequentially, the whole choicepoint will take tseq+tpar to execute. In parallel it would take only tseq, as this is the longest time for a branch. The time saved is thus tpar.



A Choicepoint example

Now all we have to do is sum the tpars and tseqs for all predicates, giving us a number for each predicate, telling us how much time we would save if we declare it as parallel. This is not entirely true though. The time is actually counted several times, as each branch of a choicepoint can include similar choicepoints. So the figures have no real absolute meaning. But they do give a good hint at where the parallelism is. Other information that are needed are the number of choicepoints and the number of branches created and whether the choicepoints have been exposed to cut or not. The number of choicepoints and branches tells us something about the granularity. The information about cut could help for better annotations of the Prolog program to reduce speculative work (speculative work is defined as the work that could turn out to be unnecessary).

3 Implementation

Stat was written in C++, just like Must, but doesn't use any graphics at all. The implementation of Stat has some similarities to Must. As the tracefiles have the same format, we read them in the same way. The trace primitives are different, though. As mentioned earlier we have four kinds of trace primitives: try, retry, remove and cut. The data is managed in two data structures: a stack and a search tree. The stack contains objects that correspond to the created choicepoints. They are created and pushed with each try and popped and destroyed with remove and cut.

Each object maintains its number of branches, its tseq and tpar etc. For each retry, the object is updated. The search tree contains a node for each predicate. When a stack-object is removed, the corresponding predicate/node is looked up in the search tree and created if not present. The predicate is then updated, so that tpar, tseq etc is accumulated through the trace. When the trace is finished, the tree is sorted in descending tpar order. The data is then presented in a table containing predicate name, tpar, tseq, tpar/choicepoint, tpar/branch, choicepoints, branches and number of cut choicepoints.

4 Stat User Manual

This section will describe how you prepare a tracefile, call Stat and interpret the output. We take for granted that the reader has some experience with UNIX and Prolog.

4.1 The Tracefile

To run stat, you must have a special Muse tracefile. The tracefile is obtained by running the Muse variant muse.seq. Muse.seq is exactly as the normal Muse except that you get sequential traces. Below is an example of a session with user input in boldface.

```
> muse.seq
...
| ?- compile(file).
{compiling file.pl...}
{file.pl compiled, 4490 msec 24387 bytes}

yes
| ?- qwe //-- solution.

solution ( 5390 ms )
yes
Do you want to dump trace [>198004 bytes] (y/n)?y
Process trace area
Dump trace to qwe_22895_000.st
Trace dumped
| ?- ^D
{ End of Muse execution. }
Exit: 22895
>
```

You dump a tracefile with the query name //-- predicate, as can be seen. The name of the tracefile will begin as specified in the query.

4.2 Starting Stat

Stat is called with

```
> stat tracefile
```

The tracefile must have been prepared as explained earlier.

4.3 The Output

Below is an example of output from Stat.

```
Reading _28817_000.st
0%                                                  100%
|---------------------------------------------------|
.....................................................
Non-cut:
```

name	tpar	tpar/cp	tpar/br	tseq	cp	branch	cut
cl/2	128604.416	128604.416	5144.176	9192.192	1	26	0
#bagof/3	67086.592	944.882	944.882	257483.008	71	142	0
generator/1	24638.464	347.021	115.674	9945.344	71	284	0
save_instances(;)/3	24556.288	345.863	345.863	203025.408	71	142	0
ins/5	293.632	5.438	4.661	252834.816	54	117	0
ins/3	223.488	10.159	3.921	72863.744	22	79	0
add23/3	151.296	2.131	2.131	353332.480	71	142	0
gen_all/3	29.440	1.132	1.132	49.152	26	52	0
run/0	1.280	1.280	1.280	137798.144	1	2	0
(null_alt)	0.000	0.000	0.000	1007.616	120	120	0

Cut:

name	tpar	tpar/cp	tpar/br	tseq	cp	branch	cut
free_variables/4	22265.088	0.144	0.144	135450279.936	154676	309352	71
bigm/3	6786.048	0.312	0.376	1401527.040	21756	39816	21756
acc23/2	1302.016	7.274	13.705	213332.224	179	274	140
lookup/6	527.872	4.399	7.650	107930.624	120	189	51
gt_cl/2	504.576	0.056	0.068	4669146.112	8956	16343	1569
proc_new(;)/11	180.992	1.508	4.022	324476.160	120	165	75
list_instances(;)/3	8.704	0.046	0.123	519184.384	191	262	120
lookup(;)/6	6.400	0.085	0.237	89322.496	75	102	48
m/3	5.632	0.000	0.001	945681.408	25120	35832	14408
bag/3	0.000	0.000	0.000	324582.912	71	71	71
member/2	0.000	0.000	0.000	89000.192	51	51	51

The predicates are divided into two groups, those that haven't been exposed to cut and those that have. In the groups, the predicates are sorted by tpar in decreasing order.

The meaning of the columns is as follows.

- name, the name of the predicate.
- tpar, a measurement of how much time that could be saved by declaring this predicate as parallel.
- tpar/cp, tpar divided by the number of choicepoints.
- tpar/br, tpar divided by the number of branches.
- tseq, a measurement of the maximum time for a branch of the predicate that could be taken.
- cp, the number of choicepoints created from this predicate.
- branch, the total number of branches in these choicepoints.
- cut, the number of choicepoints of this predicate that were exposed to cut.

5 Conclusion and Evaluation

After testing several Muse traces, we found that the conclusions you could draw from Stat output are approximately the same as when you do a conventional analysis. The results don't give much new information, but can be used as a first test before analysis, to help the developer find the potential parallel predicates quickly.

MUST

User Manual

Table of Contents

1 A Beginning Example

The best way to get started with Must is to try one example by yourself. The first thing to do is to generate a trace file. To be able to do this, we have to start the muse trace facility. Note that all the user input is in boldface.

```
> muse.trace
muse.trace \
-t 16M \
-w 12 \
booting SICStus...please wait
SICStus 0.6/Muse/sequent/trace 14.new: Thu Mar 8 14:50:28 GMT 1990
Copyright (C) 1987, Swedish Institute of Computer Science.
All rights reserved.
| ?-
```

We see that the Muse Prolog system is booted together with the trace facility. A number of default parameter settings can also be seen which are to be discussed later. Note that the default number of workers is set to 8. Now, it is time to compile the Prolog program.

```
| ?- compile(mm).
{compiling /home/bast/jan/mm.pl...}
{Info: sequential/0 not implemented}
{/home/bast/jan/mm.pl compiled, 4400 msec 24969 bytes}

yes
```

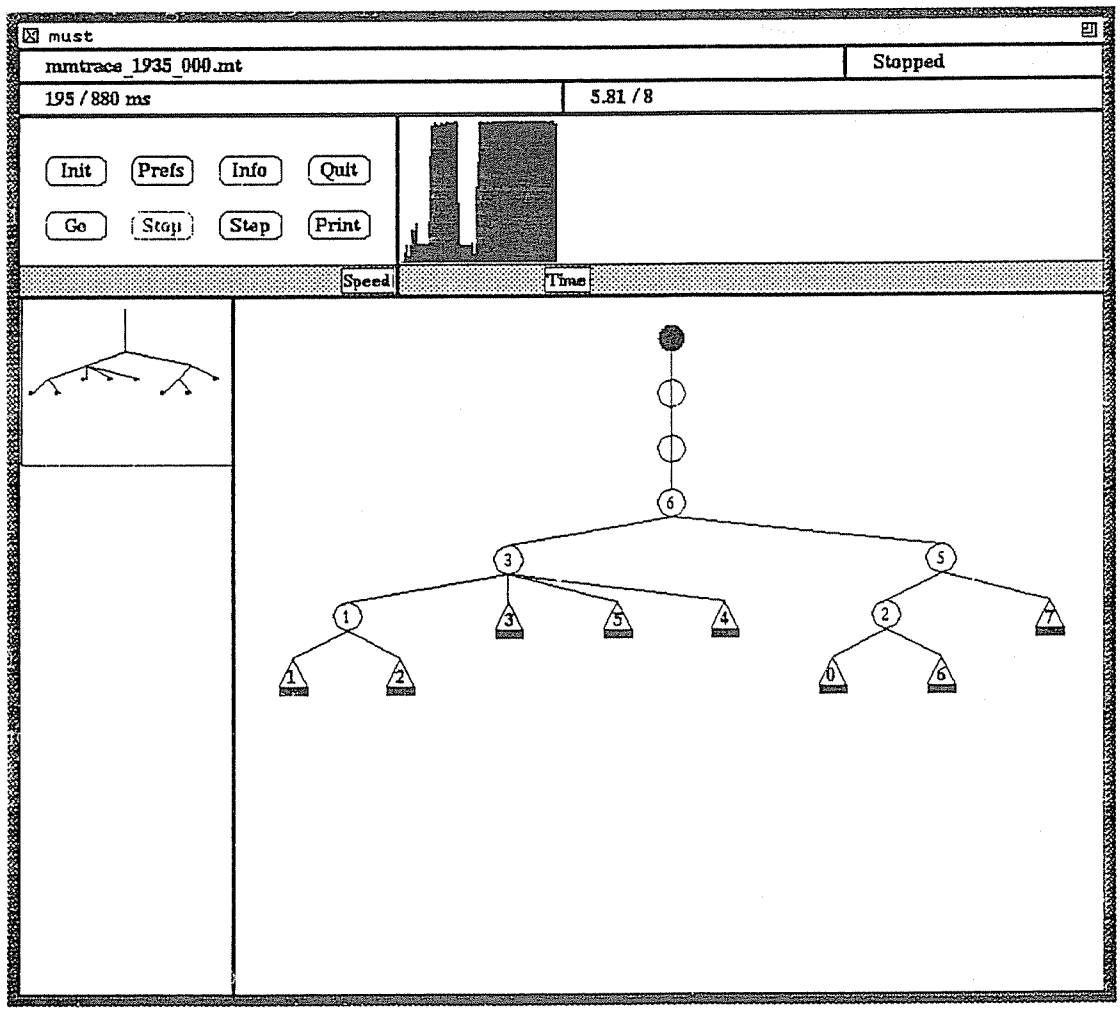
Now that the Prolog program is compiled, we are ready to generate a trace file. This is done by the query:

```
| ?- mmtrace //- mm([1,2,3,4]).
I guess [0,0,0,0]
I guess [1,1,1,1]
I guess [1,2,2,2]
I guess [1,3,2,3]
I guess [1,4,3,2]
I guess [1,2,3,4]

solution ( 640 ms )

yes
Do you want to dump trace [>15117 bytes] (y/n) y
Process trace area
Dump trace to mmtrace_1935_000.mt
Trace dumped
| ?- ^Z
Stopped
```

We see that the trace file of the query is properly generated. The next step is to start Must and load the trace file. This is easily done by giving the tracefile as an argument to Must.

```
> must mmtrace_1935_000.mt
```

The Must user interface appears on the screen with the trace file loaded. To start the trace execution, press the Go button. After a while, Must looks something like the picture below. From here you are able to, among other features, run the trace further, singlestep through it or skip back and forth by grabbing the timeslider with the middle mouse button.



A simple Must example.

This example was created to show you that the actions necessary to generate a trace file and load it into Must are very few and not so complicated as one might think. The following chapters explains all Must features along with how to generate a trace file.

2 Generating The Trace File

The trace file can only be generated with the muse Prolog tracing facility, started by issuing the command muse.trace. The muse.trace system can be run with a number of parameters, which are set by the user with switches to the muse.trace command. The most important parameters are:

- -t value. This value specifies the memory size of the trace area. The default value is 16 MBytes.
- -w number. This switch sets the number of workers executing the Prolog query. The default value is 12.

Remember that the specified value are valid through the whole tracing session. If you want to change any of these parameters, you have to restart the muse.trace system all over again, this time specifying the new parameters.To generate a trace query, simply type the line:

```
| ?- atom //-- query.
```

The atom before the double slashes and minus signs is optional and specifies the initial characters of the trace's file name.

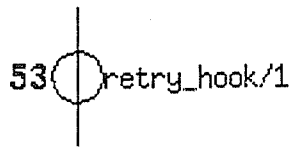
3 The Must User Interface

The Must user interface consists of a number of different element, every one with its own functions and facilities. They can be divided into three independent categories, namely the three different views, the sliders and the information fields. Here follows the complete description of each one of them.

3.1 The Mainview

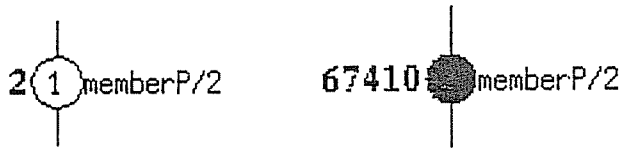
The mainview is the most essential part of the Must application. It shows (sometimes a part of) the Prolog system search tree with all necessary details. With all necessary details we mean the tree structure with the sequential and parallel nodes and the connecting lines between them, the workers moving around in the search tree, the signalling events between different workers and committed node symbols. The nodes in the tree can be of three different types, sequential or parallel shared node or unshared (local) trees. Here is a description of the different node types.

A sequential node is a node in the search tree that executes its branches sequentially. A shared sequential node is represented by a node circle with a vertical line through it. The present worker(s) are aligned to the left and the optional predicate name is aligned to the right. In the picture below workers 3 and 5 stay at the retry_hook/1 node.



A sequential node.

The next node type, the shared parallel node, executes its branches in parallel and is represented by a node circle with a number inside it. This number specifies the number of branches left to try for this node. If there are no branches left to try, the circle is filled with black. As with the sequential node, the worker(s) are aligned to the left and the optional predicate name to the right.

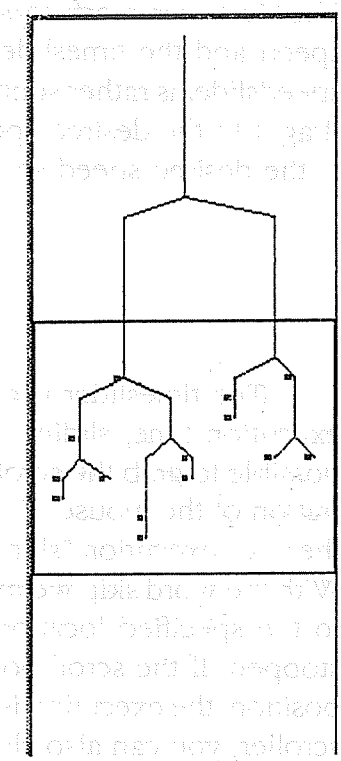


Two examples of a parallel node.

The third node type is the unshared (local) tree. It represents a local, unshared search tree not accessible for the other workers and is to be seen as a local workspace for the worker processing this node. The graphical representation of a local tree can have many shapes. A local tree with a worker executing Prolog is basically represented with a triangle with the worker number inside it. Only one worker can be present in a local tree. Any time the worker processing in the local tree changes to some non-busy mode, i e stops executing Prolog, due to the fact of some request or other reason, the triangle is replaced with a rotated square. If the worker has more than 0 unprocessed local branches (i e there is excess local work), the symbol is marked with a small black rectangle underneath it.

The four different shapes of an unshared tree.

3.2 The Sideview

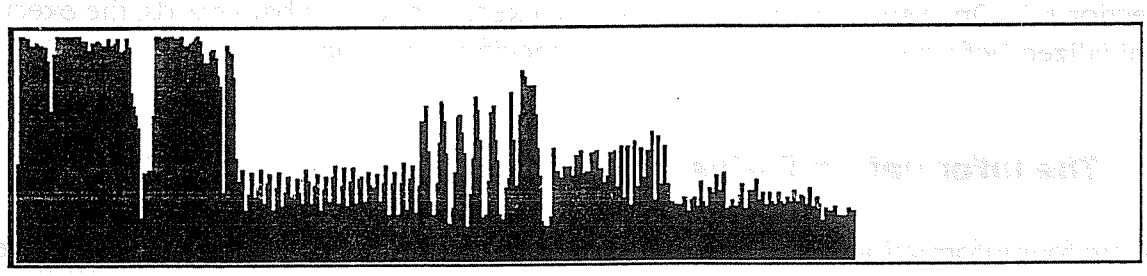


The sideview is the view to the left of the mainview. It is used to continuously showing the complete Prolog search tree without any details. The only parts of the search tree visible in the sideview are the connecting lines between the nodes and the positions of the workers, marked with a small dot. See the figure to the right. The tree visible in the sideview is a scaled version of the original tree, to be able to fit in the sideview. If the search tree should grow too big under a trace run, and by that we mean that it hits the bottom of the sideview, the whole tree is scaled down to 2/3 along the vertical axis.

Another feature in the sideview is the scroll rectangle. It continuously shows which part of the complete tree that is visible in the mainview. The scrolling rectangle can also be "grabbed", i e click inside it and then hold the mouse button down which makes the mainview scroll along with the mouse movements. You also have the possibility to click anywhere in the sideview, and make the mainview scroll immediately to that position. The scrolling feature is active even if Must is executing a trace file.

3.3 The Timeview

The timeview is the view above the mainview. It shows a graph of the number of busy workers. It consists of up to 500 black rectangles, each one with a height corresponding to the mean value of busy workers during the Prolog run time quantum it represents. The timeview is empty at the beginning of a trace execution, and a new rectangle is added as soon as the execution reaches a time mark beyond the time interval the latest rectangle represents. The maximum height of the timeview graph is a few millimeters under the upper limit of the view and corresponds to the number of workers executing the Prolog query.



A Timeview example

3.4 The Sliders

The Must user interface is equipped with two sliders, the speedslider for controlling the tracing speed and the timeslider for showing and setting the current Prolog execution time. The speedslider is rather simple, grab the scroll box of the speedslider with any mouse button and drag it to the desired speed setting. Another, faster way to reach the same result is to click at the desired speed setting point in the speedslider area.

The Speedslider

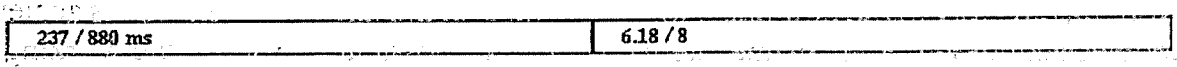
The timeslider is a little bit more powerful. It continuously shows the current Prolog execution time, sliding along right under the latest added rectangle in the timeview. It is possible to grab the scroll box of the timeslider as well, but it has to be done with the middle button of the mouse. This produces another effect than the speedslider though. This makes the trace execution "skip" very fast to the desired location, i e where the scroll box is released. With the word skip we mean that the trace execution runs without any graphic consequences to the specified location, after which the search tree is redisplayed and the execution is stopped. If the scroll box is released before the current time, i e to the left of the previous position, the execution is reinitialized before skipping to the specified location. Like the speed scroller, you can also click (with the middle button this time) at the desired time location in the timeslider. While skipping the statusfield (chapter 3.5) shows the text "Skipping to X %" where X is the relative position in the timeslider.

The Timeslider

If you should click in the timeslider with the left or right mouse button, the execution skips a fixed number of trace primitives (steps), backwards or forwards respectively. The number of trace primitives to skip is set to 1, 10, 100 or 1000 steps in the "Prefs" dialog box, see section 6.2. Once again, if the user wants the execution to skip backwards, the execution is reinitialized before skipping forward to the specified location.

3.5 The Information Fields

There are four information fields in the Must user interface. They are all positioned above the timeview and the button area. The lower leftmost field is called the time field and shows the current execution time relative to the total execution time. To the right is the speedup field. This field shows the average number of busy workers so far, i e the medium value of the timeview graph, relative to the number of workers executing the Prolog query. Both the time- and the speedup field are updated as soon as a rectangle is added in the timeview.



The Time Field and the Speedup Field

The upper leftmost field is the file name field, specifying the current trace file name. Click here to change to another trace file, or consult Section 5 for a deeper explanation. The field to the right here is the status field which shows the status of the Must execution. Examples of different status is running, stopped and skipping.

The Name Field and the Status Field

4 Starting Must

Must is started with the command Must. It can be started with a file argument which causes the specified file to be loaded into Must directly. The path to the file directory becomes the working directory. If the Must argument is a directory with an ending "/", this directory becomes the working directory in the filechooser (explained in Section 5). The following examples explain the differences.

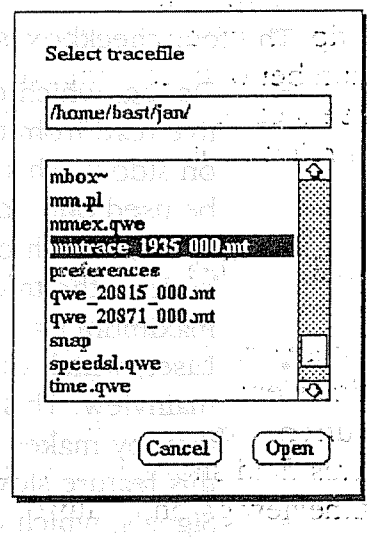
```
>must prolog/traces/mmtrace_1935_000.mt
```

This loads the trace file into Must and make prolog/traces/ the working directory.

```
>must prolog/traces/
```

This opens no trace file and makes prolog/traces/ the working directory.

5 Selecting a Trace File

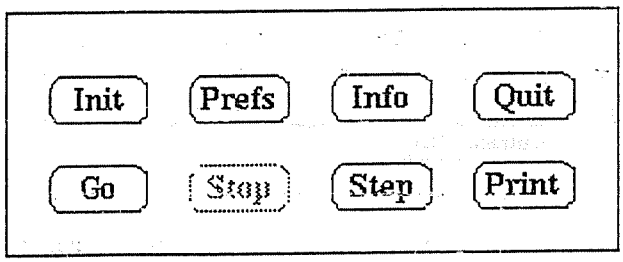


If you want to change the current trace file, click in the file name field (the upper left corner field). This pops up the file chooser dialog box displayed to the left, even if Must currently was executing a trace file. This is a scrollable list of all the files in the working directory. To scroll the file list, use the left button on the slider to the left or drag the file list with the middle button. A third alternative is to "wind" or "rewind" the file list by dragging with the right button. If you want to open a trace file, select the file by clicking on it with the left button and open it by clicking the Open button. Doubleclicking on the file name, produces the same effect. Another, more complicated way to do it is to simply write the file name in the field specifying the directory name and click the Open button.

To move down a directory, simply try to open it. This creates a new file list and changes the working directory to the new one. Moving upwards in the directory hierarchy is achieved by doubleclicking on the "../"-name alternative. A third possibility to select the working directory is to type the directory name in the directory field and click the Open button.

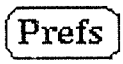
6 The Trace Control Buttons

Must is provided with eight buttons, displayed to the right, for controlling the trace execution. These buttons can be clicked with any button at any time provided that they are enabled, i e they are not "greyed out" like the disabled buttons are. A disabled button does not listen to any event.

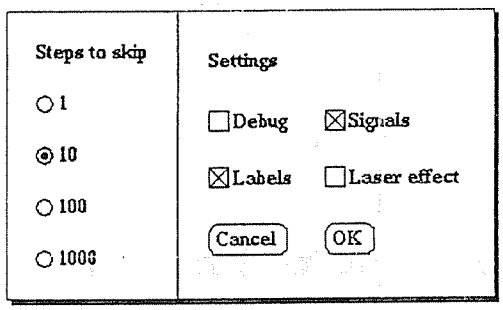


6.1 The Init Button 

The Init button causes the current execution state to be initialized, i e the initial state the execution was in when the trace file was opened. The timeslider is set to zero, the tree is deleted, only the root node (and the unshared tree under it) is displayed and the runstate is set to stopped. Use this button if you want the trace to start all over again.

6.2 The Prefs Button 

Whenever the Prefs button is pressed, the Prefs dialog box, displayed to the right, is popped up. Here you specify the user preferences, which are as follows. First, to the left, you can specify the number of steps to skip whenever the left or right button is pressed in the timeslider. Choose between 1, 10, 100 or 1000 steps by clicking the proper radio button. The default value is 10.



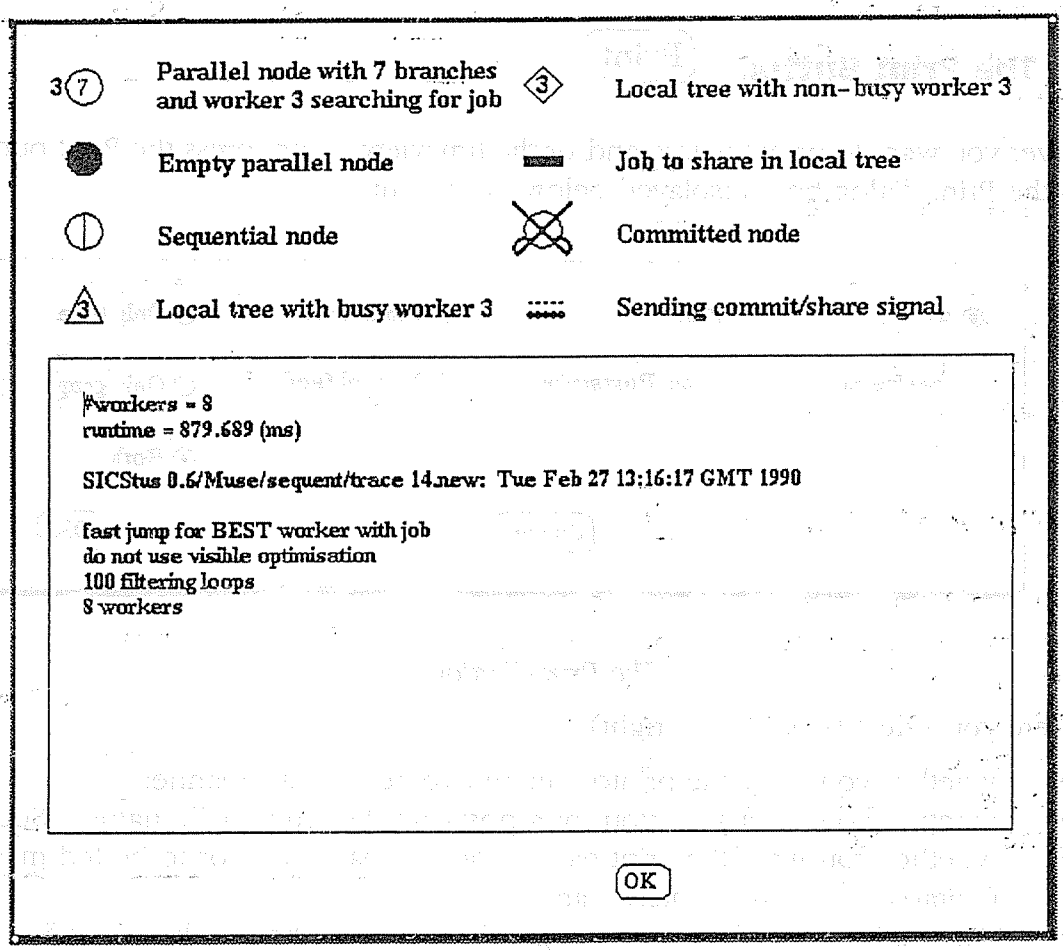
The four checkboxes to the right lets you decide about the following features.

- Debug, which causes debug printouts on stdout. This makes every trace primitive read from the trace file to be translated into an ASCII text line and printed on stdout. This slows down the maximum trace speed significantly and should be used only for debugging purposes. The default value is Off.
- Labels, which causes the predicate name to be displayed to the right of every node in the tree except for the unshared trees. This, also, slows down the maximum trace speed and therefore the default value is Off.
- Laser, which causes the workers in the tree to move in a sliding manner in the mainview. This makes the movements of all workers much more visible and thereby makes it easier to see "what goes on" in the Prolog search tree. Since this feature slows down the maximum trace speed, the default value is Off.
- Signals, which causes commit/cut and share request signals to be shown in the mainview. The signals are displayed in a "laser beam" manner with thick beams for share request signals and thin beams for commit/cut signals. Since this is an essential visual feature which cannot be seen in any other way, the default value is On.

The OK button confirms your changes and makes Must remember them until the next preferences change. The Cancel button cancels the whole operation.

6.3 The Info Button [Info]

This button pops up the info dialog box, displayed below, which contains a symbol explanation for all the symbols in the mainview. Right under the symbol explanation there is a text area displaying specific information about the current trace file.



The Info Dialog Box

6.4 The Quit Button [Quit]

The Quit button terminates the Must session.

6.5 The Go Button [Go]

The Go button starts the trace execution and executes from the current timeslider position with the current speedslider speed setting. The Must status is set to running.

6.6 The Stop Button [Stop]

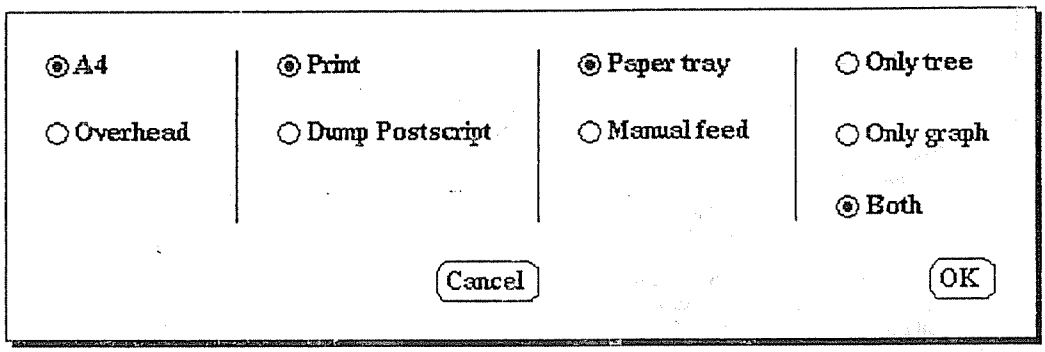
The Stop button causes the trace execution to stop and sets the Must status to stopped. If no other event comes in, the Must application process "goes to sleep" and consumes no CPU power until the next Must application event comes in.

6.7 The Step Button [Step]

The Step button causes the stopped trace execution to execute one or a few trace primitive from the trace file, depending on the Debug preference. If the Debug preference is set, only one trace primitive is executed. If not, the execution continues until a trace primitive which has a graphic consequence in the mainview occurs.

6.8 The Print Button [Print]

Whenever you want to print the tree and/or the timeview graph, press the Print button. This causes the Print dialog box, displayed below, to pop up.



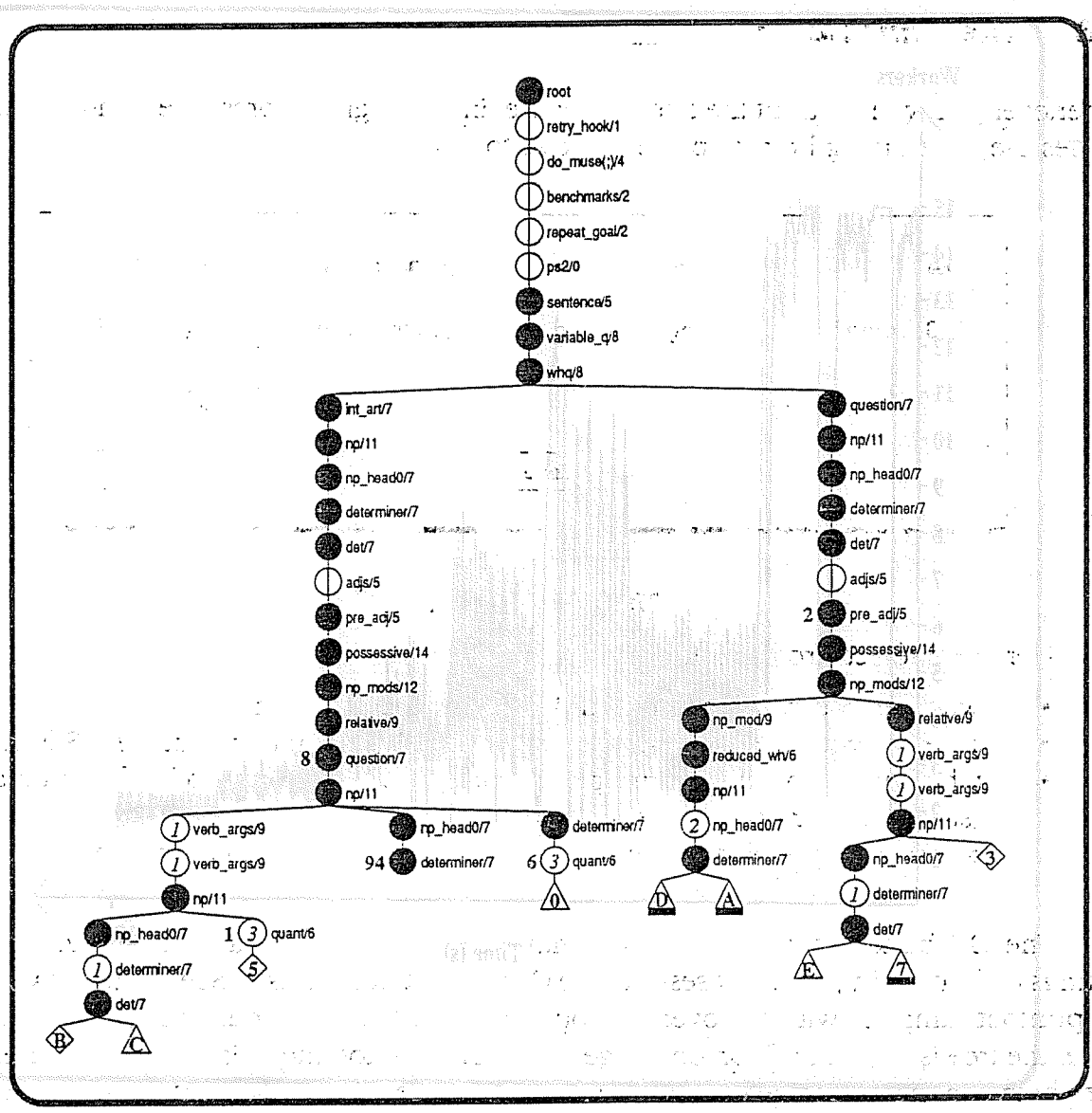
The Print Dialog Box

Here you select (from left to right):

- whether you want the printout in an overhead or A4 manner,
- whether you want a printout or a postscript file dump (file name: PSdump),
- whether you want the print paper from the paper tray or to be fed manually (suitable for transparencies), and
- whether you want the tree, the graph or both of them to be printed.

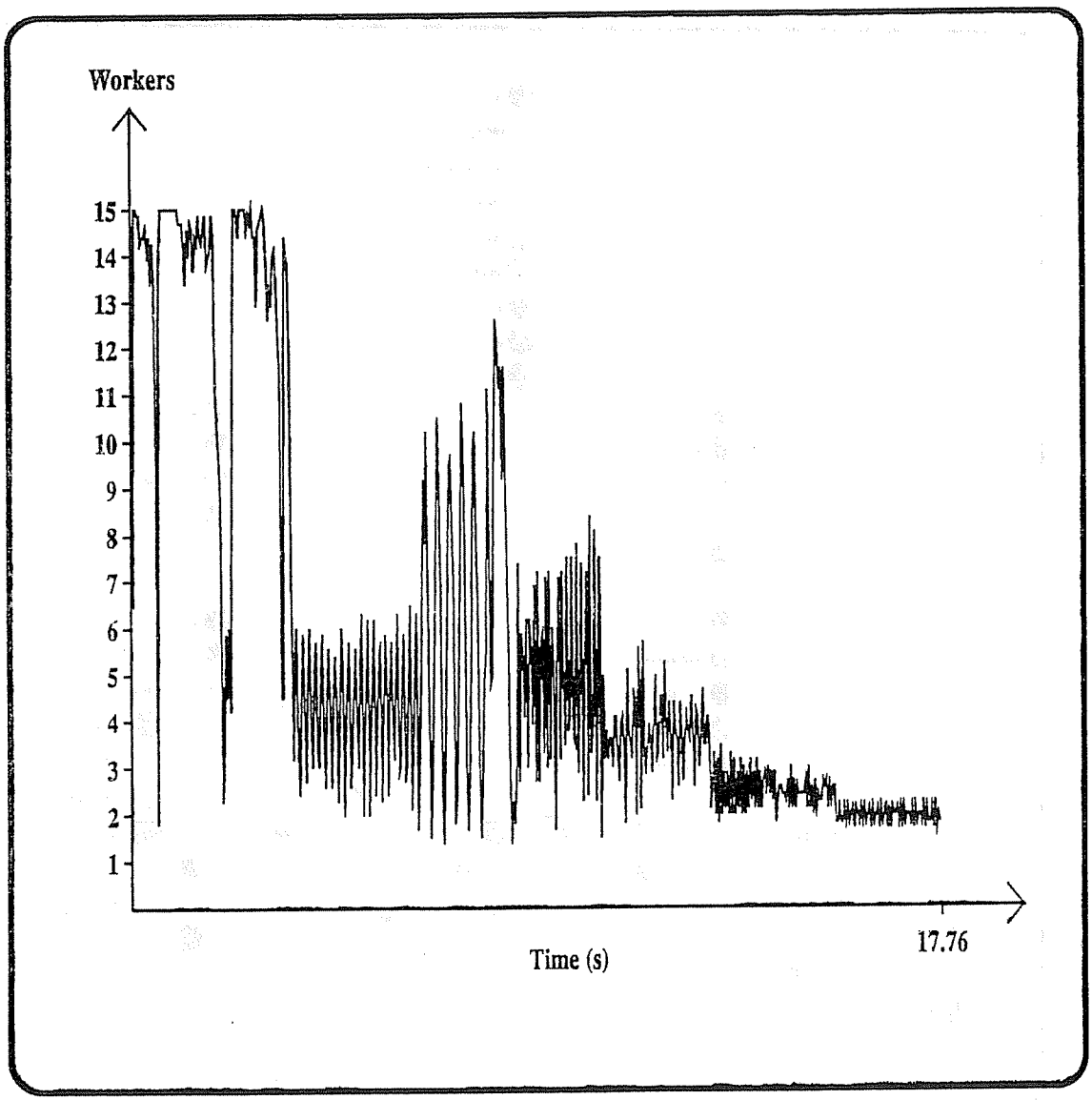
The OK button starts the printout with the current settings and makes the printing settings default during the Must session, the Cancel button cancels it. See the next two pages for printout samples (with the overhead option set). If the tree should be too big to fit one page, the tree is scaled to fit on one page. The scaling is not linear, it primarily shortens the lines between the nodes before reducing the node size. When the Print option is choosen instead of the Generate Postscript option, you should know that the printout is on the default printer. If you want the printout on another printer, choose the Generate Postscript option instead and, from a shell, issue the command:

```
>lpr -Pprinter PSdump.
```



our_b.15.mt

A Mainview Prinout Example



our_b.15.mt

A Timeview Printout Example